All You Need Is CONSTRUCT

Dominique Duval[1], Rachid Echahed[2], and Frédéric Prost[2]

[1]LJK, Univ. Grenoble Alpes and CNRS, France
[2]LIG, Univ. Grenoble Alpes and CNRS, France

Abstract

In SPARQL, the query forms SELECT and CONSTRUCT have been the subject of several studies, both theoretical and practical. However, the composition of such queries and their interweaving when forming involved nested queries has not yet received much interest in the literature. We mainly tackle the problem of composing such queries. For this purpose, we introduce a language close to SPARQL where queries can be nested at will, involving either CONSTRUCT or SELECT query forms and provide a formal semantics for it. This semantics is based on a uniform interpretation of queries. This uniformity is due to an extension of the notion of RDF graphs to include isolated items such as variables. As a key feature of this work, we show how classical SELECT queries can be easily encoded as a particular case of CONSTRUCT queries.

1 Introduction

Graph databases [8] are becoming ubiquitous in our society. The success of this recent trend in the organization of data stems from different scientific, technological and societal factors. There are different ways to encode data in terms of graphs as proposed in the literature, see e.g., RDF graphs [12] or Property graphs [9]. Various query languages can be associated to each data graph representation. In this paper, we consider the W3C standards, namely RDF [8] formalism to represent data graphs and its associated query language SPARQL [11].

An RDF graph is defined as a set of RDF triples, where an RDF triple has the form $(subject, predicate, object)$. The subject is either an IRI (Internationalized Resource Identifier) or a blank node, the predicate is an IRI and the object is either an IRI, a literal (denoting a value such as a string, a number or a date) or a blank node.

Notice that a predicate in an RDF triple cannot be a blank. For example, a triple such as $(Paul, blank_{rel}, Henry)$ standing for "there is some relationship between Paul and Henry" is not allowed in RDF, but only in generalized RDF [12, Section 7]. Following the theoretical point of view we propose in this paper, there is no harm to consider blank predicates within RDF triples. We thus consider *data graphs* in a more general setting including RDF graphs.

The language SPARQL, which is the standard query language associated to RDF, features different query forms such as SELECT or CONSTRUCT forms,

among others. Besides the W3C specifications of SPARQL [11], different authors investigated formal semantics of the language [6, 10]. The semantics associated to SPARQL queries are not uniform in general. Indeed, for instance the result of a SELECT query is a multiset of mappings [3] while the result of a CONSTRUCT query is a data graph [5]. Because of these differences between the semantics of the different forms of queries, building nested queries becomes a bit cumbersome.

However, the need of nested queries as a feature of query languages is well known [4] and nested SPARQL queries have already received some interest in the litterature. For example, in [3], nesting SELECT queries has been investigated throughoutly but CONSTRUCT queries have not been considered ; while in [7, 2], either SELECT or CONSTRUCT queries can be nested but due to the chosen semantics the FROM clause is required to nest CONSTRUCT queries.

In this paper, we consider query nesting for a core language close to SPARQL and propose a new unified semantics for the main query forms SELECT and CONSTRUCT. For this purpose, we clearly distinguish between the evaluation of a query and its result. The evaluation of query over a data graph is a set of mappings. Here a mapping should be understood algebraically as a graph homomorphism and not as a simple assignement of variables. The result of query is obtained by simply projecting the right answer as a multiset of assignements of variables or as a data graph according to the form of the query. From such semantics, one can compose queries of different forms to build involved nested subqueries.

Example 1. *To illustrate briefly our proposal, we consider Example 1 in [2] and reformulate it in our framework without using FROM clauses. In this example, one looks for emails of pairs of co-authors.*

```
SELECT ?Mail1 ?Mail2
  WHERE
    { { CONSTRUCT { ?Aut1 co-author ?Aut2. }
        WHERE
          { { ?Art bib:has-author ?Aut1 . ?Art bib:has-author ?Aut2. }
              FILTER ( !(?Aut1 = ?Aut2)) }
          }
    AND
    { ?Per1 co-author ?Per2 . ?Per1 foaf:mbox ?Mail1 .
    ?Per2 foaf:mbox ?Mail2 . }
    }
```

In order to build such a uniform semantics we had to extend the notion of RDF graphs to include isolated items. As a key feature of this work, we show how SELECT queries can be easily encoded as a particular case of CONSTRUCT queries.

The paper is organized as follows. In the next section, we introduce the main operators of a query graph algebra which are used later on when investigating the semantics of the proposed graph query language. In Section 3, a SPARQL-like language called GrAL is introduced where queries are defined as specific patterns. This language is defined by its syntax and semantics together with some illustrating examples. Concluding remarks and future work are given in Section 4.

2 The Graph Query Algebra

The Graph Query Algebra is a family of operations which are used in Section 3 for defining the evaluation of queries in the Graph Algebraic Query Language GrAL. First mappings are introduced in Section 2.1, then operations for combining sets of mappings are defined in Section 2.2. It is usual to describe the evaluation of queries in SPARQL in terms of mappings from variables to RDF terms, following [6]. In this paper, more precisely, we consider each mapping as a morphism between graphs.

2.1 Sets of mappings

Definition 1 (graph on A)**.** *For any set A and any element $t = (s, p, o)$ in A^3, the elements s, p and o are called respectively the* subject*, the* predicate *and the* object *of t. A* graph *X on A is made of a subset X_N of A called the set of* nodes *of X and a subset X_T of A^3 called the set of* triples *of X, such that the subject and the object of each triple of X is a node of X. The nodes of X which are neither a subject nor an object are called the* isolated nodes *of X. The set of* labels *of a graph X on A is the subset $A(X)$ of A made of the nodes and predicates of X.*

Remark 1. *Given two graphs X_1 and X_2 on A their* union *$X_1 \cup X_2$ is defined by $(X_1 \cup X_2)_N = (X_1)_N \cup (X_2)_N$ and $(X_1 \cup X_2)_T = (X_1)_T \cup (X_2)_T$, and similarly their* intersection *$X_1 \cap X_2$ is defined by $(X_1 \cap X_2)_N = (X_1)_N \cap (X_2)_N$ and $(X_1 \cap X_2)_T = (X_1)_T \cap (X_2)_T$. It follows that $A(X_1 \cup X_2) = A(X_1) \cup A(X_2)$ and $A(X_1 \cap X_2) = A(X_1) \cap A(X_2)$.*

Definition 2 (morphism of graphs on A)**.** *Let X and Y be graphs on a set A. A* morphism *f (of graphs on A) from X to Y, denoted $f : X \to Y$, is a partial function from $A(X)$ to $A(Y)$ which* preserves nodes *and* preserves triples*, in the following sense. Let $Dom(f)$ be the* domain of definition *of f, i.e., the subset of $A(X)$ where the partial function f is defined. Then f* preserves nodes *if $f(n) \in Y_N$ for each $n \in X_N \cap Dom(f)$ and f* preserves triples *if $f^3(t) \in Y_T$ for each $t \in X_T \cap Dom(f)^3$. Then $f_N : X_N \to Y_N$ and $f_T : X_T \to Y_T$ are the partial functions restrictions of f and f^3, respectively. Note that when n is an isolated node of X then the node $f(n)$ does not have to be isolated in Y. The* domain *of a morphism $f : X \to Y$ is X and its* range *is Y. A morphism $f : X \to Y$* fixes *a subset C of A if $f(x) = x$ for each x in $C \cap A(X)$. Then the partial function f is determined by its restriction to $A(X) \setminus C$. An* isomorphism *of graphs on A is a morphism $f : X \to Y$ of graphs on A that is invertible, which means that both $f_N : X_N \to Y_N$ and $f_T : X_T \to Y_T$ are bijections.*

Definition 3 (image)**.** *The* image *of a graph X by any partial function f from $A(X)$ to A is the graph made of the nodes $f(n)$ for $n \in X_N \cap Dom(f)$ and the triples $f^3(t)$ for $t \in X_T \cap Dom(f)^3$. It is also called the image of f and it is denoted either $Im(f)$ or $Im(X)$ when f is clear from the context. Thus each partial function f from $A(X)$ to A is a morphism of graphs on A from X to $f(X)$.*

Definition 4 (labels)**.** *From now on, the set A of labels of graphs is built from three disjoint countably infinite sets I, B and V, called respectively the sets of* resource identifiers*,* blanks *and* variables*. We denote $IB = I \cup B$, $BV = B \cup V$ and $IBV = I \cup B \cup V$. For each graph X on IBV and each subset Y of IBV, the set $A(X) \cap Y$ of labels of X which belong to Y is denoted $Y(X)$.*

Definition 5 (data and query graph, mapping)**.** Data graphs *are finite graphs on IB and* query graphs *are finite graphs on IBV. Thus each data graph can be seen as a query graph. A* mapping *m from a query graph X to a data graph Y, denoted $m : X \to Y$, is a morphism of query graphs from X to Y that fixes I.*

Remark 2. *Intuitively, the resource identifiers are the "constants", that are fixed by morphisms, while both the blanks and variables are the "variables", that may be instantiated. It is only in construct queries (Section 3.1) that blanks and variables play truly distinct roles. Thus, the precise symbol used for representing a blank or a variable does not matter: a data graph is defined "*up to blanks*" and a query graph "*up to blanks and variables*", and some care is required when several data or query graphs are in the context.*

Remark 3. *Definition 5 means that a mapping $m : X \to Y$ is a partial function from $IBV(X)$ to $IB(Y)$ that fixes I and that preserves nodes and triples. Thus, if there is a mapping from X to Y then $I(X) \subseteq I(Y)$. Each mapping $m : X \to Y$ determines a partial function $\mu : BV \to IB$, defined by $\mu(x) = m(x)$ when $x \in BV(X)$ and $\mu(x)$ is undefined when $x \in BV \setminus BV(X)$. Conversely, each partial function $\mu : BV \to IB$ can be extended as $\mu : IBV \to IB$ such that $\mu(x) = x$ for each $x \in I$; if $\mu : IBV \to IB$ preserves nodes and triples from X to Y then μ determines a mapping $m : X \to Y$, defined by $m(x) = \mu(x)$ for each $x \in IBV(X)$. In [6] and in subsequents papers like [5, 3] a* solution mapping*, or simply a* mapping*, is a partial function $\mu : V \to IB$; since it is assumed in these papers that patterns are blank-free, such mappings are related to our mappings in the same way as above, by extending μ as $\mu : IBV \to IB$ by $\mu(x) = x$ for each $x \in IB$.*

Definition 6 (set of mappings)**.** *Let X be a query graph and Y a data graph. A* set of mappings from X to Y*, denoted $\underline{m} : X \Rightarrow Y$, is a finite set of mappings $m : X \to Y$. The* domain *of $\underline{m} : X \Rightarrow Y$ is X, its* range *is Y, and its* image *is the subgraph of Y union of the images of the mappings in $\underline{m}$.*

Remark 4 (table)**.** *A set of mappings $\underline{m} : X \Rightarrow Y$ can be represented as a* table *$T(\underline{m})$ made of one line for each mapping m in $\underline{m}$ and one column for each $x \in BV(X)$, with the entry in line m and column x equal to $m(x) \in IB(Y)$ when it is defined and $\perp$ otherwise. The order of the rows and columns of $T(\underline{m})$ is arbitrary. Note that $\underline{m}$ is determined by the table $T(\underline{m})$ together with X and Y, but in general $\underline{m} : X \Rightarrow Y$ cannot be recovered from $T(\underline{m})$ alone.*

Definition 7 (compatible mappings)**.** *Two mappings $m_1 : X_1 \to Y_1$ and $m_2 : X_2 \to Y_2$ are* compatible*, written as $m_1 \sim m_2$, if $m_1(x) = m_2(x)$ for each $x \in BV(X_1) \cap BV(X_2)$. This means that for each $x \in BV(X_1) \cap BV(X_2)$, $m_1(x)$ is defined if and only if $m_2(x)$ is defined and then $m_1(x) = m_2(x)$. Given two compatible mappings $m_1 : X_1 \to Y_1$ and $m_2 : X_2 \to Y_2$, there is a unique mapping $m_1 \bowtie m_2 : X_1 \cup X_2 \to Y_1 \cup Y_2$ such that $m_1 \bowtie m_2 \sim m_1$ and*

$m_1 \bowtie m_2 \sim m_2$, *which means that* $m_1 \bowtie m_2$ *coincides with* m_1 *on* X_1 *and with* m_2 *on* X_2.

Remark 5 (About RDF and SPARQL)**.** *When dealing with RDF and SPARQL [12, 11] the set* I *is the disjoint union of the set of* IRI*s (Internationalized Resource Identifiers) and the set of* literal*s. An* RDF graph *is a set of triples on* IB*, that is, a graph on* IB *without isolated node, where all predicates are IRIs and only objects can be literals. Thus an* isomorphism *of RDF graphs, as defined in [12], is an isomorphism of graphs on* IB *as in Definition 2. The set of* RDF terms *of an RDF graph* X *is the set* $IB(X)$*. Similarly a* basic graph pattern *of SPARQL is a set of triples on* IBV *where all predicates are IRIs or variables and only objects can be literals. Thus data graphs and query graphs generalize RDF graphs and basic graph patterns, respectively.*

2.2 Operations on sets of mappings

In this Section we define some elementary transformations between sets of mappings.

Remark 6 (expressions and values)**.** *We assume the existence of a set Expr of* expressions *with subsets* $V(expr)$ *of* V *and* $B(expr)$ *of* B *for each expression* $expr$*. For each query graph* X *the* expressions on X *are the expressions* $expr$ *such that* $V(expr) \subseteq V(X)$ *and* $B(expr) \subseteq B(X)$*. We assume that there is a subset* Val *of* $I \cup \{\bot\}$ *called the set of* values *and that for each mapping* $m : X \to Y$ *and each expression* $expr$ *on* X *there is a value* $m(expr) \in Val$*. We assume that the boolean values true and false are in* Val*, as well as the numbers and strings.*

The first transformation on sets of mappings is the fundamental join operation.

Definition 8 (join)**.** *Given two sets of mappings* $\underline{m}_1 : X_1 \Rightarrow Y_1$ *and* $\underline{m}_2 : X_2 \Rightarrow Y_2$*, the* join *of* $\underline{m}_1$ *and* $\underline{m}_2$ *is the set of mappings* $Join(\underline{m}_1, \underline{m}_2) : X_1 \cup X_2 \Rightarrow Y_1 \cup Y_2$ *made of the mappings* $m_1 \bowtie m_2$ *for all compatible mappings* $m_1 \in \underline{m}_1$ *and* $m_2 \in \underline{m}_2$*:*

$Join(\underline{m}_1, \underline{m}_2) = \{m_1 \bowtie m_2 \mid m_1 \in \underline{m}_1 \wedge m_2 \in \underline{m}_2 \wedge m_1 \sim m_2\} : X_1 \cup X_2 \Rightarrow Y_1 \cup Y_2$.

Subsets of a set of mappings can be defined by a filter operation.

Definition 9 (filter)**.** *Let* $\underline{m} : X \Rightarrow Y$ *be a set of mappings and let* $expr$ *be an expression on* X*. The* filter of $\underline{m}$ by $expr$ *is the set of mappings* m *in* $\underline{m}$ *where* $m(expr) = true$*:*

$Filter(\underline{m}, expr) = \{m \mid m \in \underline{m} \wedge m(expr) = true\} : X \Rightarrow Y$.

Given a mapping $m : X \Rightarrow Y$ and a query graph X' contained in X, let $m|_{X'}$ denote the restriction of m to X'.

Definition 10 (restriction)**.** *Given a set of mappings* $\underline{m} : X \Rightarrow Y$ *and a query graph* X' *contained in* X*, the* restriction of $\underline{m}$ to X' *is the set of mappings* $Restrict_{X'}(\underline{m}) : X' \to Y$ *made of the restrictions* $m|_{X'}$ *of the mappings* m *in* $\underline{m}$ *to* X'*:*

$Restrict(\underline{m}, X') = \{m|_{X'} \mid m \in \underline{m}\} : X' \Rightarrow Y$.
Since different mappings in $\underline{m}$ may coincide on X', the number of mappings in $Restrict_{X'}(\underline{m})$ may be smaller than the number of mappings in $\underline{m}$.

Notation 1 (Notation)**.** *Given a mapping $m : X \to Y$ and a query graph X' containing X, there are several ways to extend m as $m' : X' \to Y \cup Im(X')$ where $Im(X')$ is the data graph image of X' by m'. For instance, depending on the kind of labels in $D = IBV(X') \setminus IBV(X)$:*

- *For any D, m can be extended as $m' : X' \to Y \cup Im(m')$ such that $m'(x) = x$ for each $x \in D \cap I$ and $m'(x)$ is undefined (denoted $m'(x) = \bot$) for each $x \in D \cap BV$. This is denoted:*
  $m' = Ext_{\bot}(m, X') : X' \to Y \cup Im(m')$.
- *If $D \subseteq IB$ then m can be extended as $m' : X' \to Y \cup Im(m')$ such that $m'(x) = x$ for each $x \in D \cap I$ and $m'(x)$ is a fresh blank for each $x \in D \cap B$. This is denoted:*
  $m' = Ext_{IB}(m, X') : X' \to Y \cup Im(m')$.
- *If D is made of one variable var and $expr$ is an expression on X then m can be extended as $m' : X' \to Y \cup \{m(expr)\}$ such that $m'(var) = m(expr)$. This is denoted:*
  $m' = Ext_{var \approx expr}(m, X') : X' \to Y \cup \{expr\}$.

Definition 11 (extension)**.** *Given a set of mappings $\underline{m} : X \Rightarrow Y$ and a query graph X' containing X, let $D = IBV(X') \setminus IBV(X)$. We define the following extensions of $\underline{m}$ as $\underline{m}' : X' \Rightarrow Y \cup Im(X')$ where $Im(X') = Y \cup \underline{m}'(X')$:*

- *The* extension of $\underline{m}$ to X' by undefined functions *is:*
  $Extend_{\bot}(\underline{m}, X') = \{Ext_{\bot}(m, X') \mid m \in \underline{m}\} : X' \Rightarrow Y'$.
- *If $D \subseteq IB$ then the* extension of $\underline{m}$ to X' by fresh blanks *is:*
  $Extend_{IB}(\underline{m}, X') = \{Ext_{IB}(m, X') \mid m \in \underline{m}\} : X' \Rightarrow Y'$.
- *If $D = \{var\}$ for a variable var and $expr$ is an expression on X then the* extension of $\underline{m}$ to X' by binding *var* to the values of *expr is:*
  $Extend_{var \approx expr}(\underline{m}, X') = \{Ext_{var \approx expr}(m, X') \mid m \in \underline{m}\} : X' \Rightarrow Y \cup \{expr\}$.

Note that the number of mappings in any extension of $\underline{m}$ is the same as in $\underline{m}$.

For defining the union of two sets of mappings, we first extend them by undefined functions in such a way that they both get the same domain and range.

Definition 12 (union)**.** *The* union $Union(\underline{m}_1, \underline{m}_2) : X_1 \cup X_2 \Rightarrow Y_1 \cup Y_2$ *of two sets of mappings $\underline{m}_1 : X_1 \Rightarrow Y_1$ and $\underline{m}_2 : X_2 \Rightarrow Y_2$ is the set-theoretic union of their extensions to $X_1 \cup X_2$ by undefined functions:*
$Union(\underline{m}_1, \underline{m}_2) = Extend_{\bot}(\underline{m}_1, X_1 \cup X_2) \cup Extend_{\bot}(\underline{m}_2, X_1 \cup X_2) : X_1 \cup X_2 \Rightarrow Y_1 \cup Y_2$.

Finally, we will use the well-known *projection* operation for building a multiset of mappings from a set of mappings.

Definition 13 (projection)**.** *The* projection *of a set of mappings* $\underline{m} : X \Rightarrow Y$ *to a subgraph* X' *of* X *is the* multiset *of mappings* $Project(\underline{m}, X')$ *with base set* $Restrict(\underline{m}, X') : X' \Rightarrow Y$ *and with multiplicity for each mapping* m' *the number of mappings* $m \in \underline{m}$ *such that* $m' = m|_{X'}$*. Thus the number of mappings in* $Project(\underline{m}, X')$*, counting multiplicities, is always the same as the number of mappings in* $\underline{m}$*.*

3 The Graph Algebraic Query Language

In this Section we introduce the Graph Algebraic Query Language GrAL. Its syntax and semantics for expressions and patterns are defined in a mutually recursive way: this is mainly due to the fact that expressions can be defined from patterns, using the EXISTS and NOT EXISTS syntactic blocks. Syntactically, the *queries* of GrAL are seen as patterns from the beginning: a query is a specific kind of pattern. Semantically, the *value* of a pattern over a data graph is a set of mappings (Section 3.1). In addition, when a pattern is a query then its *result* can be derived from its value: the result of a contruct-query is a data graph, the result of a select-distinct-query is a set of mappings, and the result of a select-query is a multiset of mappings, as in SPARQL (Section 3.2).

3.1 Expressions, patterns and queries

A *basic expression* is defined as usual from constants (numbers, strings, boolean values) and variables (and blanks, which act as variables here), using formal operations like $+$, $-$, *concat*, $>$, $\wedge$,... The *basic expressions on* X are defined as in Remark 6.

Definition 14 (Syntax of expressions)**.** *An* expression *expr in the language* GrAL *is either a basic expression or an expression of the form EXISTS* P_1 *or NOT EXISTS* P_1 *for some pattern* P_1*, which are expressions on* X *for every query graph* X*.*

Definition 15 (Syntax of patterns)**.** *A* pattern P *in the language* GrAL *is defined inductively as follows.*

- *A query graph is a pattern, called a* basic pattern*.*
- *If* P_1 *and* P_2 *are patterns then the following are patterns:*

  P_1 AND P_2

  P_1 UNION P_2
- *If* P_1 *is a pattern and expr an expression on* P_1 *then the following is a pattern:*

  P_1 FILTER *expr*
- *If* P_1 *is a pattern, expr an expression on* P_1 *and var a fresh variable then the following is a pattern:*

  P_1 BIND (*expr* AS *var*)

- *If P_1 is a pattern and R a query graph such that $V(R) \subseteq V(P_1)$ then the following is a pattern:*

  CONSTRUCT R WHERE P_1

- *If P_1 is a pattern and S a finite set of variables such that $S \subseteq V(P_1)$ then the following are patterns:*

  SELECT DISTINCT S WHERE P_1

  SELECT S WHERE P_1

The semantics of expressions and patterns are defined in a mutually recursive way. The value of an expression *expr* on X with respect to a set of mappings $\underline{m} : X \Rightarrow Y$ is a family $eval(\underline{m}, expr) = (m(expr))_{m \in \underline{m}}$ of elements of *Val* (Definition 16). The value of a pattern P over a data graph G is a set of mappings $[[P]]_G : [P] \Rightarrow G^{(P)}$ from a query graph $[P]$ depending only on P to a data graph $G^{(P)}$ that contains G (Definitions 18 and 19).

Definition 16 (Evaluation of expressions)**.** *The* value *of an expression expr on X with respect to a set of mappings $\underline{m} : X \Rightarrow Y$ is the family $eval(\underline{m}, expr) = (m(expr))_{m \in \underline{m}}$ of elements of Val defined as follows:*

- *If expr is a basic expression then $m(expr)$ is the given value of expr with respect to m.*
- *If $expr =$* EXISTS P_1 *then $m(expr)$ is true if there is some $m_1 \in [[P_1]]_G$ such that $m \sim m_1$ and false otherwise.*
- *If $expr =$* NOT EXISTS P_1 *then $m(expr)$ is the negation of m*(EXISTS P_1)*.*

Definition 17 (Equivalence of patterns)**.** *Two patterns are* equivalent *if they have the same value over G for every data graph G, up to a renaming of blanks.*

Definition 18 (Evaluation of non-query patterns)**.** *The* value *of a pattern P of* GrAL *over a data graph G is a set of mappings $[[P]]_G : [P] \Rightarrow G^{(P)}$ from a query graph $[P]$ depending only on P to a data graph $G^{(P)}$ that contains G. Below is the first part of the recursive definition of the value of P over G, the second part is given in Definition 19.*

- *If P is a basic pattern then $[P] = P$, $G^{(P)} = G$ and*
  $[[P]]_G : P \Rightarrow G$ *is the set of all total mappings from P (as a query graph) to G.*
- *If P_1 and P_2 are patterns then*
  $[[P_1 \text{ AND } P_2]]_G = Join([[P_1]]_G, [[P_2]]_{G^{(P_1)}}) : [P_1] \cup [P_2] \Rightarrow (G^{(P_1)})^{(P_2)}$.
- *If P_1 and P_2 are patterns then*
  $[[P_1 \text{ UNION } P_2]]_G = Union([[P_1]]_G, [[P_2]]_{G^{(P_1)}}) : [P_1] \cup [P_2] \Rightarrow (G^{(P_1)})^{(P_2)}$.
- *If P_1 is a pattern and expr an expression on P_1 then*
  $[[P_1 \text{ FILTER } expr]]_G = Filter([[P_1]]_G, expr) : [P_1] \Rightarrow G^{(P_1)}$.
- *If P_1 is a pattern, expr an expression on P_1 and var a fresh variable then*
  $[[P_1 \text{ BIND } (expr \text{ AS } var)]]_G = Extend_{var \approx expr}([[P_1]]_G, [P_1] \cup \{var\}) :$
  $[P_1] \cup \{var\} \Rightarrow G^{(P_1)} \cup \{m(expr) \mid m \in [[P_1]]_G\}$.

Definition 18 and Remark 7 are illustrated by Examples 5 to 9.

Remark 7. *Whenever* $[[P_1]]_G = G$ *then* $[[P_1$ AND $P_2]]_G$ *and* $[[P_1$ UNION $P_2]]_G$ *are symmetric in* P_1 *and* P_2*. This is the case when the pattern* P *contains no BIND, CONSTRUCT, SELECT DISTINCT or SELECT. In particular, a pattern composed of basic patterns related by ANDs is equivalent to the basic pattern union of its components. But in general the data graph* $G^{(P_1)}$ *may be strictly larger than* G*, so that the semantics of* P_1 AND P_2 *and* P_1 UNION P_2 *is not symmetric in* P_1 *and* P_2*. The semantics of patterns in* GrAL *is a set semantics: each set of mappings* $[[P_1$ UNION $P_2]]_G$ *is a set, not a multiset. However for select-queries it is possible to keep the multiplicities, as explained in Remark 11.*

The value of P_1 FILTER EXISTS P_2 *can be expressed without mentioning expressions. Indeed, it follows from Definition 18 that*

$[[P_1$ FILTER EXISTS $P_2]]_G = Restrict(Join([[P_1]]_G, [[P_2]]_{G^{(P_1)}}), [P_1])$.

In order to evaluate P_1 BIND (*expr* AS *var*) *over* G*, the fresh variable var is added to the query graph* $[P_1]$ *as an isolated node and the values* $m(expr)$ *are added to the data graph* G *as nodes, which are isolated if they are not yet nodes of* G*.*

Definition 19 (Evaluation of query patterns)**.** *Below is the second part of the recursive definition of the value of a pattern* P *of* GrAL *over a data graph* G*. The first part is given in Definition 18.*

- *If* P_1 *is a pattern and* R *a query graph such that* $V(R) \subseteq V(P_1)$ *then*
  $[[$CONSTRUCT R WHERE $P_1]]_G = Restrict(Extend_{IB}([[P_1]]_G, [P_1] \cup R), R)$:
  $R \Rightarrow G^{(P_1)} \cup Im(R)$.

- *If* P_1 *is a pattern and* S *a finite set of variables such that* $S \subseteq V(P_1)$ *then*
  $[[$SELECT DISTINCT S WHERE $P_1]]_G = Restrict([[P_1]]_G, S)$:
  $S \Rightarrow G \cup Im(S)$.

- *If* P_1 *is a pattern and* S *a finite set of variables such that* $S \subseteq V(P_1)$ *let* $Gr(S)$ *denote the query graph made of a fresh blank node* s *and a triple* (s, p_{var}, var) *for some chosen element* p_{var} *of* I *for each variable var in* S*, then*
  $[[$SELECT S WHERE $P_1]]_G = Restrict(Extend_{IB}([[P_1]]_G, [P_1] \cup Gr(S)), Gr(S))$:

  $Gr(S) \Rightarrow G \cup Im(Gr(S))$

Definition 19 and Remark 8 are illustrated by Examples 2 to 4.

Remark 8. *In order to evaluate* $Q =$ CONSTRUCT R WHERE P *over* G *one has to look for the mappings of* P *in* $[[P]]_G$*, then build a copy of* R *for each such mapping and finally merge these copies by duplicating in a suitable way the blanks of* R*. The construction of the set of mappings* $\underline{p} = [[Q]]_G : R \Rightarrow G^{(Q)}$ *from* $\underline{m} = [[P]]_G : [P] \Rightarrow G^{(P)}$ *can be described as follows. First a family of renaming functions* $(d_m)_{m \in \underline{m}}$ *is built, such that each* d_m *is an injective function from* $B(R)$ *to the set of blanks which are fresh, i.e., the blanks which are not used anywhere in the context (thus, specifically, not in* G*), and the functions* d_m *have pairwise disjoint images. For each* m *the function* d_m *is used for extending*

m as the unique mapping n on $[P] \cup R$ *such that* $n(x) = m(x)$ *for each* $x \in [P]$ *and* $n(x) = d_m(x)$ *for each* $x \in B(R)$*. Then* $\underline{n}$ *is restricted as* $\underline{p}$ *with domain* R *by restricting each* $n \in \underline{n}$ *to the subgraph* R *of* $[P] \cup R$*.*

$$\begin{array}{ccccccc} [P] & & \subseteq & & [P] \cup R & \supseteq & R \\ \underline{m}\Big\Downarrow & & Extend_{IB} & & \underline{n}\Big\Downarrow & Restrict & \underline{p}\Big\Downarrow \\ G^{(P)} & & \subseteq & & G^{(Q)} & = & G^{(Q)} \end{array}$$

Figure 1: Evaluation of a construct query.

For select-distinct queries, Definition 19 implies that

SELECT DISTINCT S WHERE $\mathrm{P}_1 \equiv$ CONSTRUCT S WHERE P_1

Indeed, the set of variables S *can be seen as a query graph made of isolated nodes, all of them variables. Then the set* $IB(S)$ *is empty and consequently* $\underline{n} = \underline{m}$*: the extension step is useless.*

For select-queries, Definition 19 implies that

SELECT S WHERE $\mathrm{P}_1 \equiv$ CONSTRUCT $Gr(S)$ WHERE P_1

The set $IB(Gr(S))$ *is non-empty: there is one blank* s *in* $Gr(S)$ *and one element* p_{var} *of* I *for each element var of* S*. It follows that* $\underline{n}$ *extends each* $m \in \underline{m}$ *with a fresh blank, image of* s*, which can be seen as an identifier for each* $m \in \underline{m}$*. When restricting* $\underline{n}$ *for computing* $\underline{p}$ *this identifier is kept, so that all mappings remain distinct.*

3.2 Queries: value and result

Definition 15 says that each query is a pattern and Definition 19 says that the value of a query is its value as a pattern, so that it is always a set of mappings, whatever the query form is. But the *result* of a query, which is defined as a by-product of its evaluation, does depend on the query form: it is a data graph for construct-queries, a set of mappings for select-distinct-queries and a multiset of mappings for select-queries.

Definition 20 (syntax of queries)**.** *A* query *in the language* GrAL *is a pattern of one of the following forms, where* P *is a pattern,* R *a query graph and* S *a finite set of variables.*

- CONSTRUCT R WHERE P
- SELECT DISTINCT S WHERE P
- SELECT S WHERE P

The value of a query Q over a data graph G in the language GrAL is its value as a pattern (Definition 19), it is the set of mappings $[[Q]]_G : [Q] \Rightarrow G^{(Q)}$. In addition, each query Q has a *result* over G, which is defined below from its value $[[Q]]_G$ in a way that depends on the form of the query.

Definition 21 (result of queries)**.** *The* result *of a query* Q *over a data graph* G *is defined from the value* $[[Q]]_G : [Q] \Rightarrow G^{(Q)}$ *as follows:*

- *If* $Q =$ CONSTRUCT R WHERE P *its result is the* data graph *image of* $[Q]$ *by* $[[Q]]_G$*:*
  $Result(Q, G) = Im([[Q]]_G)$.
- *If* $Q =$ SELECT DISTINCT S WHERE P *its result is the* set of mappings $[[Q]]_G$*:*
  $Result(Q, G) = [[Q]]_G : S \Rightarrow G^{(Q)}$.
- *If* $Q =$ SELECT S WHERE P *its result is the* multiset of mappings *projection of* $[[Q]]_G$*:*
  $Result(Q, G) = Project([[Q]]_G, S) : S \Rightarrow G^{(Q)}$.

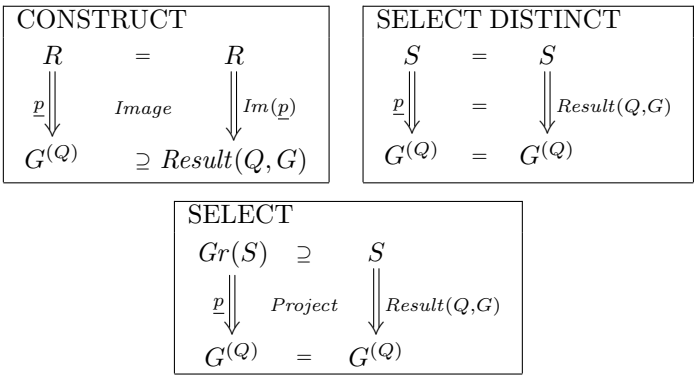


Figure 2: Result of queries.

Remark 9 (RDF and SPARQL)**.** *When Q is a construct-query and the data graph G is an RDF graph, it may happen that the data graph $Result(Q, G)$ is not an RDF graph. But the largest RDF graph included in $Result(Q, G)$ is the* answer *to Q over G in the sense of [5, Section 5]: this derives from the description of $Result(Q, G)$ in Remark 8. Using this Remark 8 we also get a description of the result of select-distinct-queries and select-queries that is the same as in [3, Section 2.3]: For select-distinct-queries, the result is the set of mappings which consists of the restrictions of all mappings in $[[P]]_G$. For select-queries, the result is the multiset of mappings with base set the restrictions of all mappings m in $[[P]]_G$, each one with multiplicity the corresponding number of m's.*

Proposition 1 (value)**.** *For any query Q with pattern P and any data graph G, the number of mappings in $[[Q]]_G$ cannot be larger than the number of mappings in $[[P]]_G$.*

Proof. Since select-queries and select-distinct-queries are equivalent to construct-queries with the same pattern, we may assume that $Q =$ CONSTRUCT R WHERE P for a pattern P and a query graph R. With the notations $\underline{m} = [[P]]_G$, $\underline{n} = Extend_{IB}(\underline{m}, [P] \cup R)$ and $\underline{p} = Restrict(\underline{n}, R)$, so that $\underline{p} = [[Q]]_G$, we know from Definitions 10 and 11 that $Card(\underline{p}) \leq Card(\underline{n}) = Card(\underline{m})$. □

Remark 10 (result)**.** *In general the value of a construct-query cannot be deduced from its result alone. However, for select-distinct-queries the value is the*

result and for select-queries the value may be recovered fom the result by choosing any fresh blanks as the images of the unique blank of $Gr(S)$.

Remark 11 (UNION and UNION ALL)**.** *The union of two multisets M_1 and M_2, respectively based on the sets X_1 and X_2, is usually defined as the multiset M based on the set $X_1 \cup X_2$ where the multiplicity of each element is the sum of its multiplicities in M_1 and M_2. When dealing with select-queries, the keyword UNION is used in* SPARQL *for the union as multisets. In* SQL *the union as multisets is obtained via the keyword UNION ALL, while UNION returns the union of the base sets. In* GrAL*, the keyword UNION always returns a set of mappings. In order to get the union as multisets of mappings we define UNION ALL as follows, with $S = V(P_1) \cup V(P_2)$:*

P_1 UNION ALL P_2 =

{SELECT S WHERE P_1} UNION {SELECT S WHERE P_2}.

See Examples 5 and 6.

Remark 12 (subqueries)**.** *Since queries are specific patterns, they can be combined at will between themselves and with other patterns, using the various syntactic building blocks for getting patterns. In particular, this provides various kinds of subqueries. See Example 8. Note that for computing the value or the result of a query, one must use the value of each subquery, not its result.*

3.3 Some examples

In the examples we assume, as in RDF, that the set I is the disjoint union of the set of IRIs and the set of literals, where the literals are strings, integers or boolean values. The literals can be combined by the usual operations on strings, integers and booleans.

We choose a concrete syntax that is similar to the syntax of SPARQL. For instance a set of triples $\{(s_1, p_1, o_1), (s_2, p_2, o_2)\}$ is written `s1 p1 o1 . s2 p2 o2 .` and braces `{ }` are used instead of parentheses ( ). The evaluation of a query Q = CONSTRUCT R WHERE P is illustrated as in Figure 1, where each set of mappings $\underline{m}$ is described by its table $T(\underline{m})$, as in Remark 4:

$$\begin{array}{ccccc} \boxed{[P]} & \subseteq & \boxed{[P] \cup R} & \supseteq & \boxed{R} \\ \| & & \| & & \| \\ \boxed{T(\underline{m})} & & \boxed{T(\underline{n})} & & \boxed{T(\underline{p})} \\ \Downarrow & & \Downarrow & & \Downarrow \\ \boxed{G^{(P)}} & \subseteq & \boxed{G^{(Q)}} & = & \boxed{G^{(Q)}} \end{array}$$

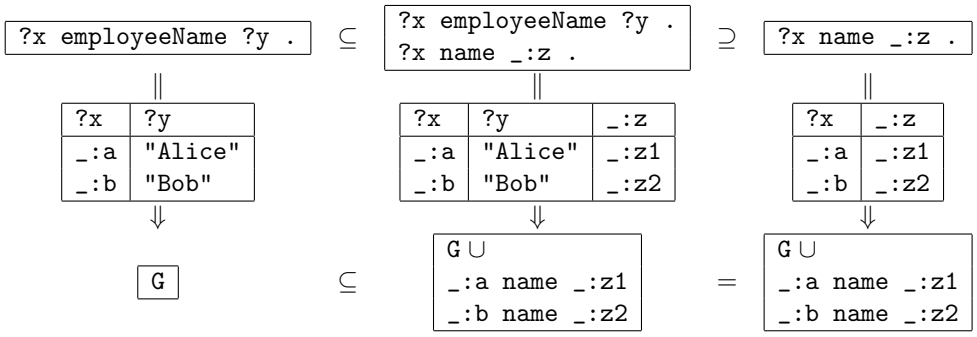
Example 2 (CONSTRUCT)**.**
This example shows how blanks are handled, whether they are in G or in R.

Data G

```
_:a employeeName "Alice" .
_:a employeeId   12345 .
_:b employeeName "Bob" .
_:b employeeId   67890 .
```

Query Q

```
CONSTRUCT { ?x name _:z }
WHERE { ?x employeeName ?y }
```



It follows that the result of Q over G is the data graph $Result(Q, G)$:

```
Result(Q,G)
_:a name _:z1 .
_:b name _:z2 .
```

Example 3 (SELECT DISTINCT)**.**
A select-distinct-query is equivalent to a construct-query.

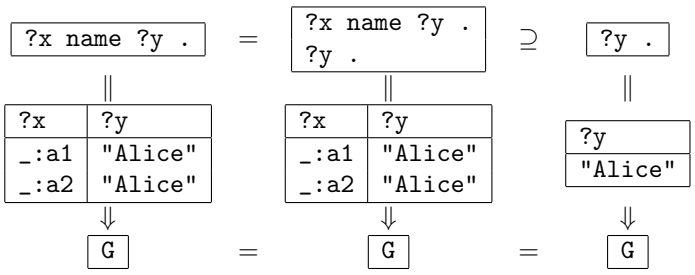
```
Data G
_:a1  name  "Alice" .
_:a1  mbox  alice@example.com .
_:a2  name  "Alice" .
_:a2  mbox  asmith@example.com .
```

```
Query Q
SELECT DISTINCT { ?y }
WHERE { ?x name ?y }
```

Equivalent construct-query:

```
Query Q1
CONSTRUCT { ?y }
WHERE { ?x name ?y }
```

The value of Q_1 over G is computed as in Example 2, it is also the value of Q over G:



It follows that the result of Q over G is the set of mappings with table:

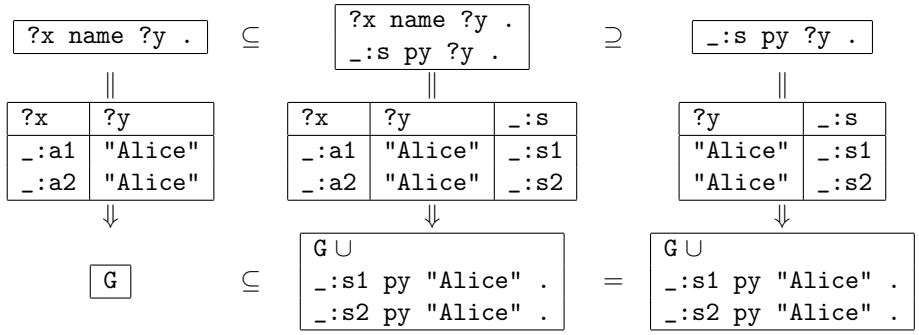
```
Result(Q,G)
?y
--------
"Alice"
```

Example 4 (SELECT)**.**
A select-query is equivalent to a construct-query, using the query graph $Gr(S)$. The data graph G is the same as in Example 3 and the query Q is a select-query, equivalent to the construct-query Q_1:

```
Query Q
SELECT ?y
WHERE { ?x name ?y }
```

```
Query Q1
CONSTRUCT { _:s py ?y }
WHERE { ?x name ?y }
```

The value of Q_1 over G is computed as in Examples 2 and 3, it is the value of Q over G:



It follows that the result of Q over G is the multiset of mappings with table:

```
Result(Q,G)
?y
--------
"Alice"
"Alice"
```

Example 5 (UNION)**.**
This example has to be compared with Example 6.

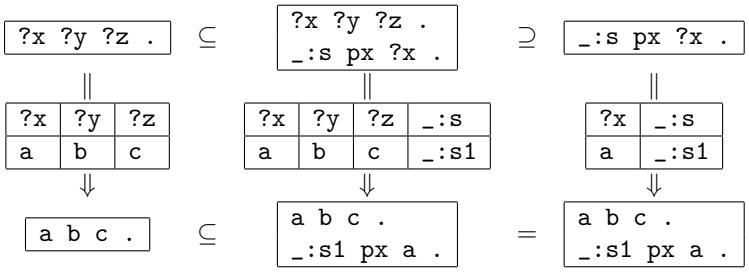
```
Data G
a b c .
```

```
Query Q
SELECT ?x
WHERE { { ?x ?y ?z } UNION { ?x ?y ?z } }
```

Definition 18 implies that P UNION $P \equiv P$ for any basic pattern P, so that here the query Q is equivalent to Q_1:

```
Query Q1
SELECT ?x
WHERE { ?x ?y ?z }
```

The evaluation of Q_1 over G runs as follows:



Thus, the result of Q over G is the multiset of mappings with table:

```
Result
?x
------
a
```

Example 6 (UNION ALL)**.**
This example has to be compared with Example 5.

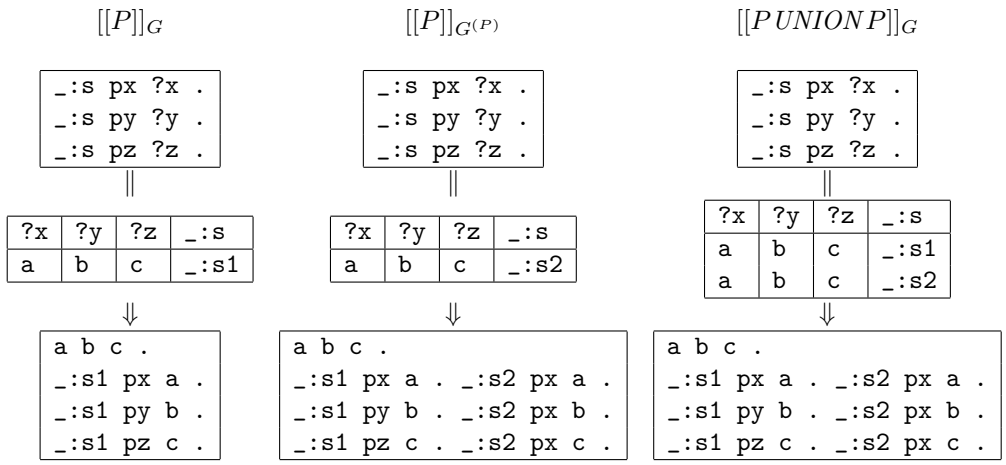
Data G
```
a b c .
```

Query Q
```
SELECT ?x
WHERE { { ?x ?y ?z } UNION ALL { ?x ?y ?z } }
```

As in Remark 11 this means that the query Q is equivalent to Q_1:

Query Q1
```
SELECT ?x
WHERE {
  { SELECT { ?x ?y ?z } WHERE { ?x ?y ?z } }
  UNION
  { SELECT { ?x ?y ?z } WHERE { ?x ?y ?z } } }
```

Here the pattern $P =$ SELECT $\{?x?y?z\}$ WHERE $\{$?x?y?z$\}$ *is not basic, and we now check that in fact P* UNION *P is not equivalent to P.*
The following diagram illustrates the value of P over G, then the value of P over $G^{(P)}$ and finally their union as sets of mappings, which is the value of P UNION *P over G. The difference between $[[P]]_G$ and $[[P]]_{G^{(P)}}$ is that the value of* `_:s` *must be a* fresh *blank, so that once some blank, say* `_:s1`*, is chosen for $[[P]]_G$ then another blank, say* `_:s2`*, must be chosen for $[[P]]_{G^{(P)}}$.*



Finally, by projecting on `?x`*, the result of Q over G is the multiset of mappings with table:*

Result

?x
a
a

Example 7 (EXISTS)**.**
This example is based on Example 4.6 in [3]. The query in [3] is similar to the query Q_0 below, however in the language GrAL *this query is not syntactically valid since $V(BOUND(?x))$ is not included in $V(\{(?y, ?y?y)\})$. A valid query Q is obtained by shifting braces.*

```
Data G
a a a .
```

```
Query Q0
SELECT ?x
WHERE
{ ?x ?x ?x
  FILTER EXISTS
  { ?y ?y ?y
    FILTER BOUND(?x)
  }
}
```

```
Query Q
SELECT ?x
WHERE
{ { ?x ?x ?x
    FILTER EXISTS { ?y ?y ?y }
  }
  FILTER BOUND(?x)
}
```

Thus $Q =$ SELECT $?x$ WHERE $\{\{$P$_1$ FILTER EXISTS P$_2\}$ FILTER BOUND $(?x)\}$ *with* $P_1 = \{(?x, ?x, ?x)\}$ *and* $P_2 = \{(?y, ?y, ?y)\}$. *The unique mapping* m_1 *from* P_1 *to* G *is such that* $m_1(?x) = a$ *and the unique mapping* m_2 *from* P_2 *to* G *is such that* $m_2(?y) = a$, *they are compatible, so that the value of* P_1 FILTER EXISTS P_2 *over* G *is* $\{m_1\}$. *Since* m_1 *binds* $?x$ *to* a, *the value of the expression* BOUND $(?x)$ *is true, thus the value of* Q *over* G *is* $[[Q]]_G = \{m_1\} : \{(?x, ?x, ?x)\} \Rightarrow \{(a, a, a)\}$.

Example 8 (subquery)**.**
This example is based on the example following Example 4.6 in [3].

```
Data G
a a a .
```

```
Query Q
SELECT ?x
WHERE
{ ?x ?x ?x
  FILTER EXISTS
  { ?y ?y ?y
    AND
    { SELECT ?x
      WHERE { ?x a ?y }
    }
  }
}
```

Here $Q =$ SELECT $?x$ WHERE P *with* $P = P_1$ *FILTER EXISTS* P_2 *and* $P_2 = P_3$ *AND* P_4 *where* P_4 *is a select-query.*

Since P_1 *is a basic pattern* $[[P_1]]_G : P_1 \Rightarrow G$ *is such that* $T([[P_1]]_G) =$

?x
a

Similarly $[[P_3]]_G : P_3 \Rightarrow G$ *is such that* $T([[P_3]]_G) =$

?y
a

The value of query P_4 *over* G *is* $[[P_4]]_G : \{(\_{:}s, p_x, ?x)\} \Rightarrow G \cup \{(\_{:}s_1, p_x, a)\}$ *such that (as in Example 4)* $T([[P_4]]_G) =$

?x	_:s
a	_:s1

Thus $[[P_2]]_G : P_2 \Rightarrow G$ *is such that* $T([[P_2]]_G) =$

?x	?y	_:s
a	a	_:s1

and finally $[[P]]_G : P_1 \Rightarrow G$ *is such that* $T([[P]]_G) =$

?x
a

When this table is seen as a multiset of mappings, it is the result of Q *over* G*:*

```
Result
?x
------
a
```

Example 9 (assignment)**.**
This example is based on Example 5.4 in [3].

Data G

```
e a b .
e c d .
f f f .
```

Query Q

```
SELECT ?x
WHERE
{ ?x a b
  FILTER EXISTS
  { ?x c d
    AND
    { ?y ?y ?y BIND ( ?y AS ?x )
    }
  }
}
```

Note that the query Q is syntactically correct since $?x \notin V(\{(?y, ?y, ?y)\})$. The main point in the evaluation of Q over G is that the subexpression EXISTS P evaluates to false, as explained below. Then clearly the result of the query is the empty multiset of mappings. The unique mapping in $[[\{(?x, c, d)\}]]_G$ sends $?x$ to e. The unique mapping in $[[\{(?y, ?y, ?y)\}]]_G$ sends $?y$ to f then BIND ($?y$ AS $?x$) extends this mapping by sending $?x$ to f. Thus the mappings are not compatible and the join is the empty set of mappings, as required.

4 Conclusion

We proposed a core language GrAL close to SPARQL for which we proposed a uniform semantics. This semantics allows one to compose different queries and patterns regardless the different forms of the queries. In this paper we did not include all SPARQL query forms such as ASK or DESCRIBE, nor did we mention aggregates or so. We intend to include such SPARQL features in a forthcoming report. The proposed framework has been illustrated on RDF graphs and SPARQL queries but it is tailored to fit any kind of graph structures with a clear notion of graph homomorphism, see e.g., the different structures mentioned in [1]. Coming back to the title of the paper, which might be a bit provocative, it emphasizes on a feature of our semantics which makes it possible to encode easily any SELECT or SELECT DISTINCT query as a CONSTRUCT query.

References

[1] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. Foundations of modern query languages for graph databases. *ACM Comput. Surv.*, 50(5):68:1–68:40, 2017.

[2] Renzo Angles and Claudio Gutiérrez. Subqueries in SPARQL. In Pablo Barceló and Val Tannen, editors, *Proceedings of the 5th Alberto Mendelzon International Workshop on Foundations of Data Management, Santiago, Chile, May 9-12, 2011*, volume 749 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2011.

[3] Mark Kaminski, Egor V. Kostylev, and Bernardo Cuenca Grau. Query nesting, assignment, and aggregation in SPARQL 1.1. *ACM Trans. Database Syst.*, 42(3):17:1–17:46, 2017.

[4] Won Kim. On optimizing an sql-like nested query. *ACM Trans. Database Syst.*, 7(3):443–469, 1982.

[5] Egor V. Kostylev, Juan L. Reutter, and Martín Ugarte. CONSTRUCT queries in SPARQL. In *18th International Conference on Database Theory, ICDT 2015, March 23-27, 2015, Brussels, Belgium*, pages 212–229, 2015.

[6] Jorge Pérez, Marcelo Arenas, and Claudio Gutiérrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, 2009.

[7] Axel Polleres, Juan L. Reutter, and Egor V. Kostylev. Nested constructs vs. sub-selects in SPARQL. In Reinhard Pichler and Altigran Soares da Silva, editors, *Proceedings of the 10th Alberto Mendelzon International Workshop on Foundations of Data Management, Panama City, Panama, May 8-10, 2016*, volume 1644 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2016.

[8] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases*. O'Reilly Media, Inc., 2013.

[9] Marko A. Rodriguez and Peter Neubauer. Constructions from dots and lines. *CoRR*, abs/1006.2361, 2010.

[10] Michael Schmidt, Michael Meier, and Georg Lausen. Foundations of SPARQL query optimization. In Luc Segoufin, editor, *Database Theory - ICDT 2010, 13th International Conference, Lausanne, Switzerland, March 23-25, 2010, Proceedings*, ACM International Conference Proceeding Series, pages 4–33. ACM, 2010.

[11] SPARQL 1.1 Query Language. W3C Recommendation, march 2013.

[12] RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation, February 2014.